# C 言語における無効なスタック領域へのポインタを検出する静的解析

矢杉 和義　五十嵐 淳

C 言語は，柔軟な低レベルメモリ処理能力を持つがゆえに，無効なメモリへのアクセスを行うバグが容易に混入する．本研究は，C 言語において起こりうるメモリアクセス違反のうち，寿命が尽きて無効になったスタック領域へのポインタへのアクセスに焦点を当て，C 言語のソースコード中に，無効なスタック領域へのポインタを生じさせうるようなコードが存在するかどうかを検査する静的解析手法を提案する．本解析は，関数定義に現れる各式の値を，その値が含むポインタの指す領域で分類し，return 文がその関数の局所変数を指すポインタを返さないことを検査する．特に，関数呼び出し式の値が指す領域を，その関数の定義を用いず実引数の情報のみから推論するので，関数定義ごとに独立に解析できる特徴がある．本研究はさらに，C 言語のサブセットとその上での本解析を形式化し，本解析により検証されたプログラムは実行しても無効なポインタが生じないことを証明する．

## 1　はじめに

C 言語は，速い実行速度と低レベルメモリ処理における柔軟な表現力をもつために，今もなおシステムプログラミング言語として幅広く用いられている．一方，C 言語にはバグを軽減する機能が殆どないため，C 言語による脆弱性被害は繰り返されている．そのため，C 言語プログラムからバグを検出する研究が求められている．

C 言語プログラムからバグを検出する手法の 1 つに静的解析がある．静的解析は，プログラムを実行することなくそのソースコードを解析することで，プログラム中に含まれるバグを検出する手法である．静的解析はプログラムの実行以前に行うので，十分な時間と計算資源を用いて解析することができるが，プログラムには実行時にしか決定できない情報も存在するので，偽陽性の発生は避けがたい．特に，型付けが弱く，柔軟な低レベルメモリ処理が可能な C 言語は，ソースコード上の記述から保証できる，実行時の値に対する不変条件が少ないために，精度の高い静的解析が困難である．そのため，先行研究 [8, 10] では，アノテーションを追加するなどの言語の拡張を行うことで，解析できる情報を増やして解析の精度を高めるものある．

本研究では，C 言語に対する，ローカル変数の寿命に関するバグを検出する静的解析手法を提案する．本解析手法には次のような特徴がある：

**言語を拡張しない**　C 言語のプログラムなら通常すでに備わっている関数宣言の情報を活用することで，アノテーションなどによる言語の拡張をすることなく解析する．

**intra-procedural な解析**　関数呼び出し式の解析を，呼び出す関数の定義に依存せず，その関数の宣言の情報のみを用いて行うため，プログラム中の各関数定義を独立に解析できる．

**線形時間アルゴリズム**　本解析は型検査の拡張によって実現される．型検査アルゴリズムはソースコードのサイズに対して線形時間のアルゴリズムなため，本解析も線形時間で実行できる．

---

Static analysis of the C language to detect pointers pointing to an invalid stack area.

Kazuyoshi Yasugi, Atsushi Igarashi, 京都大学大学院情報学研究科, Graduate School of Informatics, Kyoto University.

```
int* f(void) {
  int x = 0;
  return &x;
}
```

**Listing 1　無効なスタック領域へのポインタを返す関数**

### 1.1　論文の構成

2 節では，まず無効なスタック領域へのポインタとその問題点について説明する．3 節では，本研究が提案する静的解析手法について詳しく述べる．4 節では，C 言語のサブセット言語 Csub を定義し，Csub 上での本解析を形式化する．5 節では，Csub 上における本解析の健全性について論じる．

## 2　無効なスタック領域へのポインタ

C 言語では，関数を呼び出すたびに，その関数内で用いられるローカル変数を格納するためのスタックフレームが割り当てられる．このスタックフレームは，関数実行の終了とともに解放されるため，このスタックフレーム内のアドレスを指すポインタを呼び出し元に返すと，無効なメモリ領域を指すポインタが生じる．このポインタのことを，ここでは**無効なスタック領域へのポインタ**と呼ぶ．

Listing 1 は，無効なスタック領域へのポインタを返す関数を表している．`x` のアドレスは `f` の実行の終了とともに解放されるため，`f` の戻り値 `&x` は無効なスタック領域へのポインタとなる．無効なスタック領域へのポインタに対する間接参照は未定義動作であり [7]，このポインタの悪用は深刻なセキュリティ脆弱性につながる．したがって，Listing 1 のような無効なスタック領域へのポインタを生み出す関数定義を，事前に検出する必要がある．

Listing 1 のような単純な例では，無効なスタック領域へのポインタの発生を容易に検出できるが，関数呼び出しを介すと，問題は本質的に困難になる．

Listing 2 は関数呼び出しを介しているために解析が困難な問題の例である．`f1` は `int*` 型の引数をそのまま返す恒等関数であり，`f2` は引数によらずにヌルポインタを返す定数関数である．

- `f1` は恒等関数なので，`g1` の戻り値は `&x` である．つまり，`f1` は Listing 1 と同様に無効なスタック領域へのポインタを返す関数である．
- `f2` はヌルポインタを返す定数関数なので，`g2` の戻り値はヌルポインタである．よって，無効なポインタは生じない．

```
int* f1(int* x) {
  return x;
}
int* f2(int*) {
  return NULL;
}
int* g1(void) {
  int x = 0;
  return f1(&x);
}
int* g2(void) {
  int x = 0;
  return f2(&x);
}
```

**Listing 2　関数呼び出しを介してポインタを返す関数**

```
int* f(int* x) {
  if (x) {
    x = f(NULL);
  }
  return x;
}
int* g(void) {
  int x = 0;
  return f(&x);
}
```

**Listing 3　本解析を通らない関数**

`f1` や `f2` の実装が `g1` や `g2` と異なる翻訳単位にある場合，`g1` や `g2` の翻訳単位からは `f1` や `f2` の実際の関数定義がわからないため，`f1` や `f2` の呼び出しが安全かどうかを判定できない．

## 3　静的解析手法のアイデア

2 節で言及したように，無効なスタック領域へのポインタの存在を検査する際には，関数呼び出しの実際の挙動が呼び出し元からはわからないことが問題となる．そこで，本研究が提案する解析手法は，呼び出す関数に対しても**同じ解析を適用すること**を仮定することで，この問題を緩和するものである．

例えば，Listing 3 の `g` の解析について考える．`g` が呼び出している `f` が本解析を通ると仮定する．こ

のとき，本解析の性質から，f が返すポインタ x は，f や f がさらに呼び出しうる関数のローカル変数のアドレスではないと結論できる．つまり，そのポインタは，引数やグローバル変数から到達可能なアドレス，またはヌルポインタのいずれかに限られる．したがって，g において，f の引数に渡すポインタ &x が指している領域を把握していれば，その戻り値が指す領域の可能性を限定することができる．さらに，f の停止性も仮定すれば，たとえ f が再帰呼び出しを含んでいても，f の戻り値が指す領域を推論できる．この g の場合，f に渡しているポインタは g のローカル変数のアドレスなので，f の戻り値，ひいては g の戻り値が g のローカル変数のアドレスの可能性がある．したがって，本解析は g を危険な関数であると判断する．実際には g は無効なスタック領域へのポインタを生じさせない †1 ので，Listing 3 は偽陽性の例になっている．

### 3.1 静的解析手法の詳細

本解析は，通常の C 言語の型システムを拡張した，flow-sensitive な型システムを用いた型検査によりプログラムの解析を行う．

まず，メモリ領域の種類を，未定義な領域，現在の関数のスコープにおける自動記憶域期間をもつ領域，現在の関数の呼び出し元の関数スコープにおける自動記憶域期間をもつ領域，静的記憶域期間をもつ領域，の 4 種類に分類する．そして，これらの領域の種類に対して，u(ndef), i(nside), o(utside), g(lobal) の 4 種類の**印**を対応させる．この 4 つの印の上に，u < i < o < g から従う半順序を導入する．

C 言語の型に対して，上で導入した印の情報を追加した型を**印付き型**と呼ぶ．ポインタ型に対応する印付き型には，その型のポインタが指す領域の種類を表す印の情報を追加する．このとき，ポインタ型が印 $m$ をもつことは，その型のポインタ値が指す先の領域に対応する印が，印の上の半順序に関して $m$ 以上であることを表す．ポインタへのポインタ型の印付き型は，内側のポインタ型の印と外側のポインタ型の印の両方の情報をもつ．ポインタを含まない型に対応する印付き型は，もとの型と同じとする．そして，印付き型の上にも，印の半順序から自然に従う半順序を導入しておく．

```
int* f(void) {
  int x = 0;
  int* y = NULL;
  int** z = &y;
  y = &x;
  return *z;
}
```

**Listing 4　型システムの不変条件を壊す代入**

本解析における印付き型の型システムは，文の任意の実行時点において，各変数の印付き型とその値が次の不変条件を保つように設計されている：

- i 以上の印をもつポインタは，有効なアドレスを指すポインタであるか，またはヌルポインタである．
- o 印をもつポインタは，その時点の関数のスコープにおける自動記憶域期間をもつ領域を指さない．特に，呼び出し元の関数のスコープにおける自動記憶域期間をもつ領域を指してもよい．
- g 印をもつポインタは，自動記憶域期間をもつ領域を指さない．特に，呼び出し元の関数のスコープにおける自動記憶域期間をもつ領域も指さない．

型システムがこの不変条件を満たすならば，全ての return 文が i 以下の印をもつポインタ値を返さなければ，関数から無効なスタック領域を指すポインタが返されることはないことを保証できる．本解析が行う型システムの型検査アルゴリズムは，基本的にはそれぞれの式や文がこの不変条件を満たすことを確認しながら，代入文や制御構文の分岐の合流のたびに型環境を更新するものである．

ここからは，本解析における型検査アルゴリズムのうち，特筆すべき処理について，具体例を交えながら説明する．

代入文の型検査について，Listing 4 を例に考える．f において，y はまず g 印をもつヌルポインタで初期化される．z はローカル変数 y のアドレスで初期化さ

†1　g はヌルポインタを返す．

```
void f(int n) {
  int* x = &n;
  int* y = &n;
  int** z = n ? &x : &y;
  *z = NULL;
}
```

**Listing 5　間接参照による代入**

れるので，z は g 印の int 型ポインタを指す i 印のポインタ型をもつ．その次の文では y に対して，i 印のポインタ型をもつローカル変数 x のアドレスが代入される．もしこの代入を認めてしまった場合，*z は g 印のポインタであるが，*z の値は y の値つまりローカル変数 x のアドレスである．x は自動記憶域期間をもつ変数なので，型システムの不変条件を満たさない．これを防ぐため，本解析では右辺式の印が左辺式の印未満のときの代入は，型検査エラーとなるように設計した．

左辺式が間接参照を含む代入文の型検査について，Listing 5 を例に考える．f において，x も y も i 印のポインタである．z は n の値に依存して，x か y のいずれかのアドレスで初期化されるが，いずれにせよ z は i 印のポインタ型への i 印のポインタ型をもつ．ここで，*z に対する g 印のヌルポインタの代入は，上述の代入文における印の制約 (左辺式の印 i ≤ 右辺式の印 g) を満たすが，実際に x と y のどちらにヌルポインタが代入されるのかわからないため，x と y のどちらの印付き型を更新すべきなのかわからない．このように左辺式が間接参照を含む場合，一般に印付き型を更新するべき変数を特定することができない．そのため本解析では，間接参照の先に対する代入の場合には，印付き型の更新を行わない．印付き型の更新を行わなくても，前述の代入文における印の制約さえ満たしていれば，型システムの不変条件は保たれる．

C 言語の関数においては，引数に渡したポインタを経由して関数の出力を行う技法がしばしば用いられるが，Listing 6 に示すような場合には，型システムの不変条件が壊れる可能性がある．f は第 1 引数が指す先の int* 型の領域に第 2 引数のポインタ値を書き込む関数である．g において，x は g 印のヌルポインタで初期化される．g における f の呼び出しは，

```
void f(int** x, int* y) {
  *x = y;
}
void g(void) {
  int* x = NULL;
  int y = 0;
  f(&x, &y);
}
```

**Listing 6　引数経由のポインタ値の書き込み**

```
typedef struct {
  int *a, *b;
} S;
int* f(short* n, int** x, S* y);
int* g(int* x) {
  short n = 0;
  S y = {x, NULL};
  return f(&n, &x, &y);
}
```

**Listing 7　関数呼び出し式の印付き型の推論**

x = &y という代入文と等価であるので，x はローカル変数 y を指す．しかし，型システムの側からは，x は依然として g 印のポインタであるので，型システムの不変条件を満たさない．この問題は，f の側の型システムからは，f の引数のポインタがどこを指しているのかわからない点にある．そのため，f における代入文 *x = y は，x と y に引き渡されるポインタ値の印によっては型システムの不変条件を壊す代入となる．そこで本解析では，引数のポインタ値の先の領域に対する g 未満の印付き型をもつ値の代入は，型検査エラーとなるように設計した．g の印付き型をもつ値の代入であれば，型システムの不変条件を壊すことはないので認めてよい．

関数呼び出し式の型検査について，Listing 7 を例に考える．構造体 S は int* 型の値を 2 つ保持する型である．f の戻り値型は int* なので，f の宣言のみから考えられる f が返しうるポインタは，

1. 第 2 引数 x が指す先のポインタ値，
2. 第 3 引数 y が指す先の構造体 S がもつ 2 つのポインタ値，
3. グローバル変数から到達可能なポインタ値，
4. ヌルポインタ

のいずれかであると考えられる．特に，ポインタ間

キャストは型システムの不変条件を壊すため，本解析はこれを認めてみない．そのため，ここでは short* から int* へのキャストは考慮しない．前述の代入文における印の制約により，グローバル変数にはローカル変数のアドレスは含まれていないので，3, 4 はいずれも g 印のポインタ値と考えてよい．1, 2 のポインタの印は，g における f の呼び出しの実引数から知ることができ，順に o と o, g であることがわかる．つまり，f の戻り値の印は o 以上であることがわかる．よって，f の呼び出し式の印付き型には o 印を与えればよい．この推論により，呼び出す関数の実装に依存することなく解析を行うことができる．

以上のアイデアをふまえた，本解析が行う flow-sensitive な型システムによる型検査アルゴリズムを次にまとめる：

**変数宣言**

- 未初期化のポインタ値の印付き型には u をつける．
- 仮引数のポインタ値の印付き型には o をつける．

**アドレス式**

- 自動記憶域期間をもつ変数のアドレス値の印付き型には i をつける．
- 自動記憶域期間をもたない変数のアドレス値およびヌルポインタの印付き型には g をつける．

**関数呼び出し式**

- 関数呼び出し式の戻り値型が基本型の場合，その呼び出し式の印付き型は，戻り値型と対応する印付き型のうち，次の条件を満たす最大のものとする：実引数に含まれる任意の値に対して，その値の型が戻り値型と同じならば，呼び出し式の印付き型はその値の印付き型以下である．
- 関数呼び出し式の戻り値型が複合型の場合，その型に含まれる各基本型に与える印を上の規則にしたがって導いたのち，それらの基本型を結合した印付き型を，呼び出し式の印付き型とする．

**キャスト式**

- ポインタ型への，またはポインタ型からのキャスト式は，型検査エラーとなる．

**代入文**

- 右辺値の印付き型が左辺値の印付き型未満のとき，型検査エラーとなる．
- 左辺式がポインタの間接参照を含む場合，左辺値の印付き型は更新しない．左辺式がポインタの間接参照を含まない場合，左辺値の印付き型を右辺値の印付き型で更新する．
- 左辺式が o 印のポインタ値に対する間接参照を含み，かつ左辺値がポインタ値を含むとき，型検査エラーとなる．

```
void f(int** x, int* y) {
  *x = y;
}
int* g(void) {
  int* x = NULL;
  f(&x, NULL);
  return x;
}
```

**Listing 8　代入文における印の制約の違反の例**

**制御文**

- 分岐が合流するとき，各変数の合流後の印付き型を，次の条件を満たす印付き型のうち最大のもので更新する：任意の分岐に対して，合流後の印付き型は，その分岐におけるその変数の印付き型以下である．

**return 文**

- return する値の印付き型が i 以下の印を含むとき，型検査エラーとなる．

### 3.2　偽陽性

他の静的解析手法と同様に，本解析にも偽陽性が存在する．Listing 8, 9, 10, 11 に挙げたプログラムは，いずれも本解析の検査を通らないが，C 言語プログラムとして合法なもの，すなわち偽陽性と判断されるプログラムの例である．Listing 8, 9 のような偽陽性は，本解析に特有のものであり，型検査アルゴリズムの設計選択の結果である．一方，Listing 10, 11 のような偽陽性は，本手法に限らず静的解析では本質的に解決困難な問題である．

**代入文における印の制約の違反 (Listing 8)**

3.1 節で述べたように，本解析の型システムの不変条件を壊す代入文は，たとえ合法な操作であっても本解析の検査を通らない．

```
int* f(int* x, int* y) {
  return y;
}
int* g(void) {
  int x = 0;
  return f(&x, NULL);
}
```

**Listing 9　印付き型の推論の誤差の例**

```
char a[4] = "abc";
char* f(void) {
  intptr_t x = (intptr_t)a;
  ++x;
  return (char*)x;
}
```

**Listing 10　ポインタ型に関するキャストの例**

```
int* f(void) {
  int x = 0;
  if (0) {
    return &x;
  } else {
    return NULL;
  }
}
```

**Listing 11　到達不能な実行パスの例**

**印付き型の推論の誤差 (Listing 9)**

本解析は，関数呼び出し式の印付き型を，その関数の実装に依存することなく決定する．そのため，関数の実装によっては，印付き型と実際の戻り値との間に誤差が生じ，合法なプログラムであっても本解析の検査を通らないということが起こりうる．この問題は，戻り値型と同じ型の引数が複数ある場合に顕著になる．また，前述の Listing 3 もこの種の偽陽性である．

**ポインタ型に関するキャスト (Listing 10)**

ポインタ型から異なるポインタ型へのキャスト，あるいはポインタ型と整数型の間のキャストは，型システムを壊すものであるから，たとえ合法な操作であっても，型システムに依存した本解析の検査を通らない．

**到達不能な実行パス (Listing 11)**

到達不能な実行パスにおいて違法な操作が行われた場合，実際にはその違法な操作は実行されることはない．しかし静的解析では，一般に実行パスの到達可能性を決定不能であるため，型検査は到達可能性にかかわらず行う必要がある．そのため，到達不能な実行パスにおける違法な操作に対して本解析の検査が通らないために，違法なプログラムであると判定される可能性がある．

## 4　形式化

3 節で説明した静的解析手法のアイデアが健全であること，すなわち，本解析を通過したプログラムには，無効なスタック領域へのポインタに対するアクセスが含まれないことを保証するためには，本解析を形式化し，健全性を証明する必要がある．本節では本解析の形式化を行う．4.1 節では，解析の対象とする C 言語のサブセット言語 Csub の構文を定義する．4.2 節では，Csub の操作的意味論を定義する．4.3 節では，もとの C 言語と Csub との差分について取り上げ，その妥当性を検討する．4.4 節では，本解析のアイデアを形式化した Csub の型システムを定義する．なお，Csub の構文および操作的意味論の定義においては，Clight [3] の形式化を参考にしている．

### 4.1　Csub の構文

C 言語のサブセットを形式化した Csub の構文を図 1 に示す．オリジナルの C 言語全体に対する形式化は，その規模のために困難であるので，C 言語の代わりに Csub のプログラムを，形式化された本解析の対象とする．

**型**　型は `int` 型と `ptr` 型と `pair` 型の 3 種類である．`int` 型は整数型，`ptr` 型はポインタ型を表しており，`pair` 型は `int` 型または `ptr` 型からなる列を表す．`pair` 型を用いて構造体を模倣する．

**式**　式は左辺値式と右辺値式のいずれかである．式は型と値を持ち，左辺値式は対応する領域をもつ．

**左辺値式**　`.0` 演算子と `.1` 演算子は，`pair` 型の左辺値式から，それぞれ前者と後者を取り出す演算を表す．`*` 演算子は，ポインタ型の式からそのポインタが指す先の領域を取り出す演算を表す．

**右辺値式**　`+` 演算子は，整数型に対する加算を表す．`&` 演算子は，左辺値式からその領域を指すポイン

| | |
|---|---|
| 識別子 | $x$ |
| 整数 | $n$ |
| 型 | $\tau ::= \sigma \mid \texttt{pair}(\sigma, \tau)$ |
| | $\sigma ::= \texttt{int} \mid \texttt{ptr}(\tau)$ |
| 式 | $e ::= l \mid r$ |
| 左辺値式 | $l ::= x \mid l\texttt{.0} \mid l\texttt{.1} \mid \texttt{*}e$ |
| 右辺値式 | $r ::= n \mid e \texttt{ + } e \mid \texttt{\&}l$ |
| 文 | $s ::= \texttt{skip} \mid \texttt{return}\ e \mid$ |
| | $l \texttt{ = } e \mid l \texttt{ = } x\texttt{(}e\texttt{)} \mid s\texttt{;}\ s \mid$ |
| | $\texttt{if(}e\texttt{)\{}s\texttt{\}else\{}s\texttt{\}} \mid \texttt{while(}e\texttt{)\{}s\texttt{\}}$ |
| 変数宣言 | $d ::= \tau\ x$ |
| | $d^+ ::= d \mid d,\ d^+$ |
| 関数定義 | $f ::= \tau\ x\texttt{(}d\texttt{)\{}d^+\texttt{;}\ s\texttt{\}}$ |
| | $f^+ ::= f \mid f,\ f^+$ |
| プログラム | $P ::= d^+,\ f^+$ |

**図 1　Csub の構文**

タを作る演算を表す.

**文**　$l$ = $e$ は代入文を表す．$l$ = $x$(e) は，関数呼び出し $x$(e) の戻り値を $l$ に代入する代入文を表す.

**変数宣言**　$\tau\ x$ は，$\tau$ 型の変数 $x$ の宣言を表す.

**関数定義**　$\tau\ x$(d){$d^+$; $s$} は，引数宣言が $d$，戻り値型が $\tau$ である関数 $x$ の定義を表す．$d^+$ は関数 $x$ のローカル変数の宣言を，$s$ は関数 $x$ の本体を表す.

**プログラム**　$d^+$, $f^+$ は，グローバル変数宣言の列 $d^+$ と，関数定義の列 $f^+$ からなるプログラムを表す.

### 4.2　Csub の操作的意味論

Csub の操作的意味論の判断で用いるメタ変数を図 2 に示す.

**メモリ状態**　$M$ は Csub のメモリモデルを数学的に表現したオブジェクトであり，直観的にはいくつかのメモリブロックの集合である．メモリ状態の詳細な扱いに関しては，Csub で採用するメモリモデルとともに後述する.

**メモリブロック・オフセット・アドレス**　メモリブロック $b$ は有限のバイト列を表す．メモリブロック中の各バイトは整数値のオフセット $\delta$ を用いて特定する．すなわち，メモリ上のアドレス $p$ は，メモリブロック $b$ とオフセット $\delta$ の組によって表現される.

| | |
|---|---|
| メモリ状態 | $M$ |
| メモリブロック | $b$ |
| オフセット | $\delta$ |
| アドレス | $p ::= (b, \delta)$ |
| 値 | $v ::= u \mid \mathrm{cons}(u, v)$ |
| | $u ::= \mathrm{integer}(n) \mid \mathrm{pointer}(p) \mid \mathrm{undef}$ |
| 出力 | $\omega ::= \mathrm{pass} \mid \mathrm{ret}(v)$ |
| ローカル環境 | $E\colon x \mapsto b$ |
| グローバル環境 | $G\colon x \mapsto b$ |
| 関数環境 | $F\colon x \mapsto f$ |

**図 2　Csub の操作的意味論におけるメタ変数**

**値**　integer($n$) は整数値 $n$ を表す．pointer($p$) はメモリ上のアドレス $p$ を指すポインタ値を表す．undef は未初期化の値を表す．cons($u, v$) は pair 型の式と対応しており，整数値，ポインタ値または未初期化値の一連の列を表す.

**出力**　$\omega$ は関数定義中の文の実行の状態を表す．pass は関数を実行中であることを表し，ret($v$) は値 $v$ を return する文を実行したあとであることを表す.

**ローカル環境**　$E$ は，ローカル変数 $x$ と，その変数が割り当てられているメモリブロック $b$ との対応を表す写像である.

**グローバル環境**　$G$ は，グローバル変数 $x$ と，その変数が割り当てられているメモリブロック $b$ との対応を表す写像である.

**関数環境**　$F$ は関数名 $x$ とその名をもつ関数定義 $f$ との対応を表す写像である.

Csub におけるメモリモデルの表現には，[9] におけるメモリモデルを採用している．[9] では，メモリ状態 $M$ に対して，次の 4 つの操作を定義している．これらの関数は，いずれも戻り値はオプション型であり，成功すれば結果が $\lfloor \cdot \rfloor$ に囲まれて出力され，失敗すれば $\varepsilon$ が出力される.

| | |
|---|---|
| 左辺値式 | $G, E \vdash_{\mathrm{L}} l, M \Downarrow p$ |
| 式 | $G, E \vdash_{\mathrm{R}} e, M \Downarrow v$ |
| 値格納 | $G, E \vdash_{\mathrm{V}} l, v, M \Downarrow M'$ |
| 文 | $F, G, E \vdash_{\mathrm{S}} s, M \Downarrow \omega, M'$ |
| 関数 | $F, G \vdash_{\mathrm{F}} x(v), M \Downarrow v', M'$ |
| プログラム | $G \vdash_{\mathrm{P}} P, M \Downarrow v$ |

**図 3　Csub の操作的意味論の判断**

$\texttt{alloc}(M, lo, hi) = \lfloor (b, M') \rfloor$：メモリブロックを追加する．メモリ状態 $M$，下位オフセット $lo$，上位オフセット $hi$ を入力として，$lo$ を下界，$hi$ を上界とする新しいメモリブロック $b$ と，新しいメモリ状態 $b$ の組を出力する．

$\texttt{free}(M, b) = \lfloor M' \rfloor$：メモリブロックを解放する．メモリ状態 $M$，メモリブロック $b$ を入力として，そのメモリブロックを解放した後の新しいメモリ状態 $M'$ を出力する．

$\texttt{load}(t, M, b, \delta) = \lfloor v \rfloor$：メモリから値を読み出す．メモリ型 $t$，メモリ状態 $M$，メモリブロック $b$，オフセット $\delta$ を入力として，メモリブロック $b$ とオフセット $\delta$ によって指定されるアドレスから，メモリ型 $t$ のサイズ分の値 $v$ を読み出して出力する．

$\texttt{store}(t, M, b, \delta, v) = \lfloor M' \rfloor$：メモリへ値を書き込む．メモリ型 $t$，メモリ状態 $M$，メモリブロック $b$，オフセット $\delta$，値 $v$ を入力として，メモリブロック $b$ とオフセット $\delta$ によって指定されるアドレスに，値 $v$ を書き込んだ新しいメモリ状態 $M'$ を出力する．

Csub では，これらの操作を primitive な演算として用いることで，メモリモデルの詳細な実装を隠蔽し，より高級な操作的意味論の構築に注力している．また，メモリは 4 バイト境界でアラインメントし，integer($n$), pointer($p$), undef にはいずれも 4 バイトの領域を使用する．

続いて，Csub の操作的意味論の判断を図 3 に示す．

**左辺値式**　グローバル環境 $G$，ローカル環境 $E$，メモリ状態 $M$ のもとで，左辺値式 $l$ はアドレス $p$ に評価される．左辺値式は式でもあるので，式の判断とは明示的に区別する必要がある．

**式**　グローバル環境 $G$，ローカル環境 $E$，メモリ状態 $M$ のもとで，式 $e$ は値 $v$ に評価される．

**値格納**　グローバル環境 $G$，ローカル環境 $E$，メモリ状態 $M$ のもとで，左辺値式 $l$ を評価した先のアドレスに値 $v$ を格納した，新しいメモリ状態 $M'$ に評価される．

**文**　関数環境 $F$，グローバル環境 $G$，ローカル環境 $E$，メモリ状態 $M$ のもとで，文 $s$ を実行した結果の出力 $\omega$ と新しいメモリ状態 $M'$ に評価される．出力が pass だった場合，後続の文が引き続き評価されるが，出力が $\mathrm{ret}(v)$ だった場合，後続の文は評価されない．

**関数**　関数環境 $F$，グローバル環境 $G$，メモリ状態 $M$ のもとで，関数 $x$ の引数に値 $v$ を渡して呼び出し，その戻り値 $v'$ と新しいメモリ状態 $M'$ に評価される．

**プログラム**　グローバル環境 $G$，メモリ状態 $M$ のもとで，グローバル変数 $G$ が適切に初期化されているならば，`main` 関数の戻り値 $v$ に評価される．

Csub では，左辺値式および式の評価は副作用を引き起こさない．

Csub の操作的意味論の判断は，実行時エラーに直面した場合に error に評価される．後の 4.3 節で詳しく述べるように，操作的意味論の判断の評価における error は，そのまま無効なスタック領域を指すポインタへの間接参照を表すように定義される．導出途中の判断で発生した error は伝播し，最終的にプログラムの判断の結果が error となるように導出規則が定義される．

Csub における操作的意味論の導出規則の抜粋を図 4 に示す．

**(S-Store)**　左辺値式に対して値を格納する判断の導出規則である．メモリ状態 $M_0$ 上のアドレス $(b, \delta)$ に対して，primitive 演算 `store` を用いて，値 $u$ を設定した新しいメモリ状態 $M_1$ に評価される．

**(S-Call)**　関数呼び出しの戻り値を左辺値式に代入する判断の導出規則である．まず，式 $e$ を評価して値 $v_0$ を取り出し，$v_0$ を $x$ に与えた関数呼

$$\frac{G,\, E \vdash_{\mathrm{L}} l,\, M_0 \Downarrow (b, \delta) \qquad \texttt{store}(\texttt{int32}, M_0, b, \delta, u) = \lfloor M_1 \rfloor}{G,\, E \vdash_{\mathrm{V}} l,\, u,\, M_0 \Downarrow M_1}\ (\text{S-Store})$$

$$\frac{G,\, E \vdash_{\mathrm{R}} e,\, M_0 \Downarrow v_0 \qquad F,\, G \vdash_{\mathrm{F}} x(v_0),\, M_0 \Downarrow v_1,\, M_1 \qquad G,\, E \vdash_{\mathrm{V}} l,\, v_1,\, M_1 \Downarrow M_2}{F,\, G,\, E \vdash_{\mathrm{S}} l = x(e),\, M_0 \Downarrow \mathrm{pass},\, M_2}\ (\text{S-Call})$$

$$\frac{G,\, E \vdash_{\mathrm{R}} e,\, M \Downarrow v}{F,\, G,\, E \vdash_{\mathrm{S}} \texttt{return}\ e,\, M \Downarrow \mathrm{ret}(v),\, M}\ (\text{S-Ret})$$

$$\frac{F,\, G,\, E \vdash_{\mathrm{S}} s_0,\, M_0 \Downarrow \mathrm{ret}(v),\, M_1}{F,\, G,\, E \vdash_{\mathrm{S}} s_0 \texttt{;}\ s_1,\, M_0 \Downarrow \mathrm{ret}(v),\, M_1}\ (\text{S-SeqRet})$$

$$\frac{G,\, E \vdash_{\mathrm{L}} l,\, M \Downarrow (b, \delta) \qquad \texttt{load}(\texttt{int32}, M, b, \delta) = \varepsilon}{G,\, E \vdash_{\mathrm{R}} l,\, M \Downarrow \mathrm{error}}\ (\text{E-Load})$$

**図 4　Csub の操作的意味論**

び出しを評価して，関数の戻り値 $v_1$ と新しいメモリ状態 $M_1$ 得る．$l$, $v_1$, $M_1$ に対する値格納の判断を評価することで，代入後のメモリ状態 $M_2$ を得る．

**(S-Ret)** return 文の実行を表す判断の導出規則である．式 $e$ を評価した値 $v$ に対して，$\mathrm{ret}(v)$ に評価される．

**(S-SeqRet)** 連続する文の実行を表す判断の導出規則である．前半の文 $s_0$ を実行して出力 $\mathrm{ret}(v)$ を得た場合，後半の文 $s_1$ は実行されない．連続する文全体の判断における出力と新しいメモリ状態は，前半の文の判断と同じものが使われる．

**(E-Load)** 左辺値式からの値の読み込み，つまり lvalue-to-rvalue conversion が失敗したことを表す判断の導出規則である．まず，$l$ を評価してアドレス $(b, \delta)$ を得る．そのアドレスから `load` を用いて 4 バイトのメモリを読み出そうとするが，`load` が無効値 $\varepsilon$ を返して失敗したため，error に評価される．

### 4.3　C 言語と Csub との差異

Csub の操作的意味論におけるエラーは，無効なスタック領域を指すポインタの間接参照によってのみ起こるように設計されている．つまり，ポインタを間接参照した結果は，次のいずれかに限られる：

- 有効なポインタの間接参照であり，ポインタが指す先の領域を取得できる．
- ポインタが指す先は解放済みのスタック領域であり，そのポインタの間接参照はエラーを引き起こす．

この設計により，Csub は無効なスタック領域を指すポインタのみに注目した体系となっている．以下では，もとの C 言語にあって，Csub にはない言語要素を，Csub が含まない理由について述べる：

**ヌルポインタ**　ヌルポインタは無効なポインタ値である．よって，もし Csub にヌルポインタを導入すると，無効なスタック領域を指すポインタのみに注目した設計を壊すことになる．そのため，Csub はヌルポインタを含まない．

**構造体**　Csub が構造体そのものを持たず，`pair` 型によって構造体を模倣する設計を選択しているのは，単に体系の簡潔のためである．

`pair` 型には構造体のようにタグ名がついていないため，自分自身の構造体へのポインタをメンバにもつ構造体を表現できない．一方，Csub にはヌルポインタがないため，このような自己参照的な構造を初期化することができない．そのため，Csub は自己参照的な構造を導入していない．

`pair` 型は `int` 型や `ptr` 型からなる一列の構造しか成すことができないため，他の構造体をメンバにもつ構造体のような型を表現できない．しかし，入れ子になった構造体を潰して 1 つの構造体に変換する操作を考えると，`pair` 型は構造体と同程度の表現力をもつといえる．

**整数値に対する算術演算**　Csub には，整数値に対する算術演算として `+` しか含まれていないが，この制限は単に体系の簡潔のためである．本解析はポインタの指す領域のみに着目しているため，整数値に対するその他の算術演算を Csub に含めても，本解析は影響を受けない．

**ポインタ値に対する算術演算**　もし Csub においてポインタ値に対する算術演算を行うことができれば，メモリブロックの範囲外を指すような無効なポインタ値を表現できるようになり，無効なスタック領域を指すポインタ値のみに注目した設計を壊すことになる．そのため，Csub はポインタ値に対する算術演算を認めていない．また，この帰結として，Csub は配列を含まない．

**共用体・キャスト**　共用体やキャストは型システムを壊すことができる．本解析はプログラムを静的解析のみによって検査するため，型システムが壊されると何も保証できなくなってしまう．そのため，Csub は共用体やキャストを含まない．

**関数**　C 言語における関数呼び出しは式であるが，Csub では式から副作用を取り除くために，代入と組み合わせた文として扱っている．

Csub の関数の引数は，体系の簡潔のためにちょうど 1 つに限られている．`void` 型の引数や戻り値をもつ関数は，ダミーの `int` 型の値を受け渡しする関数を対応させることができ，複数の引数を受け取る関数は，`pair` を用いて引数を 1 つにまとめた関数を対応させることができるので，Csub の関数は C 言語の関数と同程度の表現力をもつといえる．

Csub には，体系の簡潔のために関数ポインタが存在せず，関数呼び出しは常に関数名を直接用いる必要がある．しかし，もし Csub に関数ポインタを導入したとしても，関数ポインタは静的記憶域期間をもつので，関数ポインタに印 g をつけるようにすれば，容易に本解析を拡張することができると考える．

**制御構文**　Csub は体系の簡潔のために，制御構文として `if` と `while` しかもたない．しかし，本解析の型検査アルゴリズムは制御構文の形式に依存していないので，例えば `switch`，`break` や `goto` などのその他の制御構文を Csub に加えても，分岐の合流における型環境の更新をうまく定義することで，容易に本解析を拡張することができる．

**変数宣言**　関数定義の先頭だけでなく，任意のブロックの先頭でのローカル変数宣言を許容すると，Listing 12 のような，ローカルスコープ内における無効なスタック領域へのポインタが発生しうる．これに対しても本解析を適用するためには，印に加えて，ブロックの深さの情報も同時に扱う必要があり，体系を簡潔に保つことができない．そのため，Csub では関数定義の先頭以外の位置でのローカル変数宣言を認めていない．

```
void f(void) {
  int* x = NULL;
  {
    int y = 0;
    x = &y;
  }
  *x = 0;  // NG
}
```

**Listing 12　ローカルスコープ内における無効なスタック領域へのポインタ**

$$
\begin{aligned}
\text{印}\quad & m ::= \mathrm{u} \mid \mathrm{i} \mid \mathrm{o} \mid \mathrm{g} \\
\text{印付き型}\quad & \mu ::= \nu \mid \mathrm{pair}(\nu, \mu) \\
& \nu ::= \mathrm{int} \mid \mathrm{ptr}(\mu, m) \\
\text{グローバル型環境}\quad & \Delta\colon x \mapsto \mu \\
\text{ローカル型環境}\quad & \Gamma\colon x \mapsto \mu
\end{aligned}
$$

**図 5　Csub の型判断におけるメタ変数**

### 4.4　型システム

Csub の型判断で用いるメタ変数を図 5 に示す．

**印・印付き型**　Csub の型は，3 節において扱ったものと同じ印付き型である．

**グローバル型環境**　$\Delta$ は，グローバル変数 $x$ と，その印付き型 $\mu$ との対応を表す写像である．

**ローカル型環境**　$\Gamma$ は，ローカル変数 $x$ と，その印付き型 $\mu$ との対応を表す写像である．

続いて，Csub の型判断を図 6 に示す．

**左辺値式**　グローバル型環境 $\Delta$，ローカル型環境 $\Gamma$ のもとで，左辺値式 $l$ が印付き型 $\mu$ をもち，$l$ のアドレスの印が $m$ であることを表す．

**式**　グローバル型環境 $\Delta$，ローカル型環境 $\Gamma$ のもとで，式 $e$ が印付き型 $\mu$ をもつことを表す．

**値格納**　グローバル型環境 $\Delta$，ローカル型環境 $\Gamma$

| | |
|---|---|
| 左辺値式 | $\Delta, \Gamma \vdash_{\mathrm{L}} l \colon (\mu, m)$ |
| 式 | $\Delta, \Gamma \vdash_{\mathrm{R}} e \colon \mu$ |
| 値格納 | $\Delta, \Gamma \vdash_{\mathrm{V}} l, \mu \Rightarrow \Gamma'$ |
| 文 | $F, \Delta, \Gamma \vdash_{\mathrm{S}} s \colon \tau \Rightarrow \Gamma'$ |
| 関数定義 | $F, \Delta \vdash_{\mathrm{F}} f$ |
| プログラム | $\vdash_{\mathrm{P}} P$ |

**図 6　Csub の型判断**

$$\frac{}{\mathrm{int} \sqsubseteq \mathrm{int}}$$

$$\frac{\mu_0 \sqsubseteq \mu_1 \qquad m_0 \leq m_1}{\mathrm{ptr}(\mu_0, m_0) \sqsubseteq \mathrm{ptr}(\mu_1, m_1)}$$

$$\frac{\nu_0 \sqsubseteq \nu_1 \qquad \mu_0 \sqsubseteq \mu_1}{\mathrm{pair}(\nu_0, \mu_0) \sqsubseteq \mathrm{pair}(\nu_1, \mu_1)}$$

**図 7　代入可能な印付き型の対応を表す二項関係**

のもとで，左辺値式 $l$ に対して印 $\mu$ をもつ値を格納した結果，新しいローカル環境 $\Gamma'$ が得られることを表す．

**文**　関数環境 $F$，グローバル型環境 $\Delta$，ローカル型環境 $\Gamma$ のもとで，戻り値型 $\tau$ である関数定義の中の文 $s$ を実行した結果，新しいローカル環境 $\Gamma'$ が得られることを表す．

**関数定義**　関数環境 $F$，グローバル型環境 $\Delta$ のもとで，関数 $f$ の本体の文が正しく型付けされ，関数 $f$ の本体中の全ての return 文が適切な印付き型の値を返していることを表す．

**プログラム**　プログラム $P$ 中の全ての関数定義が正しく型付けされることを表す．

Csub の型判断の導出規則において用いる二項関係の定義を図 7 に表す．二項関係 $\mu_0 \sqsubseteq \mu_1$ は，$\mu_0$ の印付き型をもつ左辺値式に対して $\mu_1$ の印付き型をもつ右辺値式を代入可能であることを表す．ポインタ型の間の二項関係 $\mathrm{ptr}(\mu_0, m_0) \sqsubseteq \mathrm{ptr}(\mu_1, m_1)$ の導出において，$m_0 \leq m_1$ を要求している部分が，3.1 節において説明した，型システムの不変条件を保つための，代入文における印の制約を表現している．

Csub の型判断の導出規則の抜粋を図 8 に示す．$|\mu|$ は，$\mu$ から全ての印を取り除いた型を表す．$\mathrm{fill}(\tau, m)$ は，$\tau$ に対応する印付き型のうち，その印の全てが $m$ であるものを表す．

$$\frac{\Gamma_0(x) \sqsubseteq \mu \qquad \Gamma_0[\mu/x] = \Gamma_1}{\Delta, \Gamma_0 \vdash_{\mathrm{V}} x, \mu \Rightarrow \Gamma_1}\ (\text{T-StoreLocal})$$

$$\frac{\Delta, \Gamma \vdash_{\mathrm{R}} e \colon \mu \qquad \Delta, \Gamma_0 \vdash_{\mathrm{V}} l, \mu \Rightarrow \Gamma_1}{F, \Delta, \Gamma_0 \vdash_{\mathrm{S}} l = e \colon \tau \Rightarrow \Gamma_1}\ (\text{T-Assign})$$

$$\frac{\Delta, \Gamma \vdash_{\mathrm{R}} e \colon \mu \qquad |\mu| = \tau \qquad \mathrm{fill}(\tau, \mathrm{o}) \sqsubseteq \mu}{\Gamma \vdash_{\mathrm{S}} \texttt{return}\ e \colon \tau \Rightarrow \Gamma}\ (\text{T-Ret})$$

**図 8　Csub の型判断の導出規則**

**(T-StoreLocal)**　ローカル変数への値の格納の型判断を導出する．変数 $x$ のもとの印付き型 $\Gamma_0(x)$ に対して $\mu$ の印付き型をもつ値が格納可能であれば，$\Gamma_0$ の $x$ に対する対応を $\mu$ で更新し，新しい型環境 $\Gamma_1$ を得る．

**(T-Assign)**　代入文の型判断を導出する．式 $e$ の印付き型 $\mu$ を取得し，左辺値式 $l$ に対して $\mu$ の印付き型をもつ値を格納する．格納可能であれば，更新された新しい環境 $\Gamma_1$ を得る．

**(T-Ret)**　return 文の型判断を導出する．return する値 $e$ の印付き型 $\mu$ を取得し，$\mu$ に対応する印無し型が現在の関数定義の戻り値型 $\tau$ と一致することを確認する．$\mu$ の印付き型をもつ値が $\mathrm{fill}(\tau, o)$ の印付き型をもつ領域に格納可能であるとき，つまり，$\mu$ が i や u の印を含まないとき，式 $e$ の値を return できる．

## 5　性質と証明

4 節における解析手法の形式化が完了したので，解析手法の健全性について述べることができるようになった．

**定理** (健全性)．任意のプログラム $P$，グローバル環境 $G$，メモリ状態 $M$ に対して，$\vdash_{\mathrm{P}} P$ かつ，ある $v$ が存在して $G \vdash_{\mathrm{P}} P, M \Downarrow v$ が成り立つならば，$v \neq \mathrm{error}$ が成り立つ．

$\vdash_{\mathrm{P}} P$ とは，プログラム $P$ が正しく型付けされる，すなわち $P$ が本解析の検査を通ることを表す．

$G \vdash_{\mathrm{P}} P, M \Downarrow v$ とは，適切に初期化されたグローバル環境 $G$ とメモリ状態 $M$ のもとで，プログラム $P$ を実行すると，プログラムが停止して，出力として $v$ が得られることを表す．プログラムが正常に実行できれば $v$ は何らかの有効な値になるが，途中で無効なスタック領域へのポインタに対する間接参照が存在した場合，error が伝播して出力される．したがって，この健全性定理の主張は，本解析の検査を通過する Csub のプログラムには，無効なスタック領域へのポインタに対する間接参照が存在しないことを表す．

この健全性の証明が，本研究の目標である．現時点では，この証明は完了していない．

## 6 関連研究

C 言語プログラムのバグを静的解析によって検出する研究は，これまでにも数多く行われている．それらの中でも，Cyclone [8] と CCured [10] は本研究のアプローチに近く，大いに参考にした．

Cyclone は，C 言語のポインタに Region-based Memory Management [11] を導入した C 言語方言である．Cyclone のポインタには，region と呼ばれる，そのポインタが指す領域を識別するアノテーションが付随している．Cyclone の型システムは，ポインタに付加された region の情報を用いて，ぶら下がりポインタなどのバグを検出することができる．region の情報は関数宣言におけるポインタ引数にも指定されるため，関数をまたいだ型検査が可能である．

CCured は，C 言語のポインタに対して 3 種類のアノテーションを付加することで，ポインタの機能を細分化した C 言語方言である．3 種類のポインタはそれぞれ異なる用途に合わせた異なる性質をもっている．ポインタの種類ごとに，その性質に合わせた実行時検査コードを挿入することで，ポインタに対する操作が正しいことを，型検査と実行時検査の両面から確認する．

C 言語方言に対する静的解析手法は，既存のプログラムに対して適用することが難しい欠点があるが，強固な理論的裏付けをもつために，解析を通過したプログラムにある種のバグが存在しないことが保証できる．また，静的解析単体だけではなく，動的検査と組み合わせて偽陽性の少ない解析手法にするなど，柔軟に設計の選択を行う余地が存在する．

一方，C 言語を拡張せずにそのまま解析する静的解析器の研究も盛んに行われている．この種の静的解析器は，現実のプログラムに対してそのまま適用できる利点をもつが，C 言語の性質上，バグの種類によっては検出精度の向上が困難な場合も多い．そのため，検出できるバグの種類とその精度が異なるさまざまな静的解析器 [1, 2, 4–6] が提案されている．

## 7 おわりに

本研究は，C 言語プログラムにおける，無効なスタック領域へのポインタに対する間接参照に起因するバグを検査する，静的解析手法を開発した．そして，この静的解析手法の健全性を確かめるために，C 言語のサブセット言語 Csub を定義した．

今後の課題としては，本解析手法の健全性証明と実装が挙げられる．Csub に対する本解析手法の形式化は，本解析が健全であってはじめて意味をもつため，健全性証明の完遂は急務である．そして，本解析の実装を用いて現実の C 言語プログラムを解析することで，本解析における偽陽性の程度を具体的に評価する必要があると考えている．

また，本解析手法は自動記憶域期間に起因するバグという，限定的な問題に対してしか対処することができない．そのため，今後はヒープ領域におけるぶら下がりポインタなど，より多くのバグに対処できる静的解析手法へと拡張する必要があると考えている．

## 参 考 文 献


[1] Ball, T. and Rajamani, S. K.: The SLAM Project: Debugging System Software via Static Analysis, *ACM SIGPLAN Notices*, Vol. 37, No. 1(2002), pp. 1–3.

[2] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R.: The software model checker Blast, *International Journal on Software Tools for Technology Transfer*, Vol. 9, No. 5(2007), pp. 505–525.

[3] Blazy, S. and Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language, *Journal of Automated Reasoning*, Vol. 43, No. 3(2009), pp. 263–288.

[4] Calcagno, C. and Distefano, D.: Infer: An automatic program verifier for memory safety of C

programs, *NASA Formal Methods*, (2011), pp. 459–465.

[ 5 ] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X.: The ASTREÉ Analyzer, *European Symposium on Programming*, 2005, pp. 21–30.

[ 6 ] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B.: Frama-C, *Software Engineering and Formal Methods*, 2012, pp. 233–247.

[ 7 ] International Organization for Standardization: *ISO/IEC 9899:2011, Programming languages – C*, 2011.

[ 8 ] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y.: Cyclone: A Safe Dialect of C., *USENIX Annual Technical Conference*, 2002, pp. 275–288.

[ 9 ] Leroy, X. and Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations, *Journal of Automated Reasoning*, Vol. 41, No. 1(2008), pp. 1–31.

[10] Necula, G. C., McPeak, S., and Weimer, W.: CCured: Type-safe retrofitting of legacy code, *ACM SIGPLAN Notices*, Vol. 37, No. 1(2002), pp. 128–139.

[11] Tofte, M. and Talpin, J. P.: Region-Based Memory Management, *Information and Computation*, Vol. 132, No. 2(1997), pp. 109–176.